

EMS Scripting with Jython



Jerzy M. Nogiec

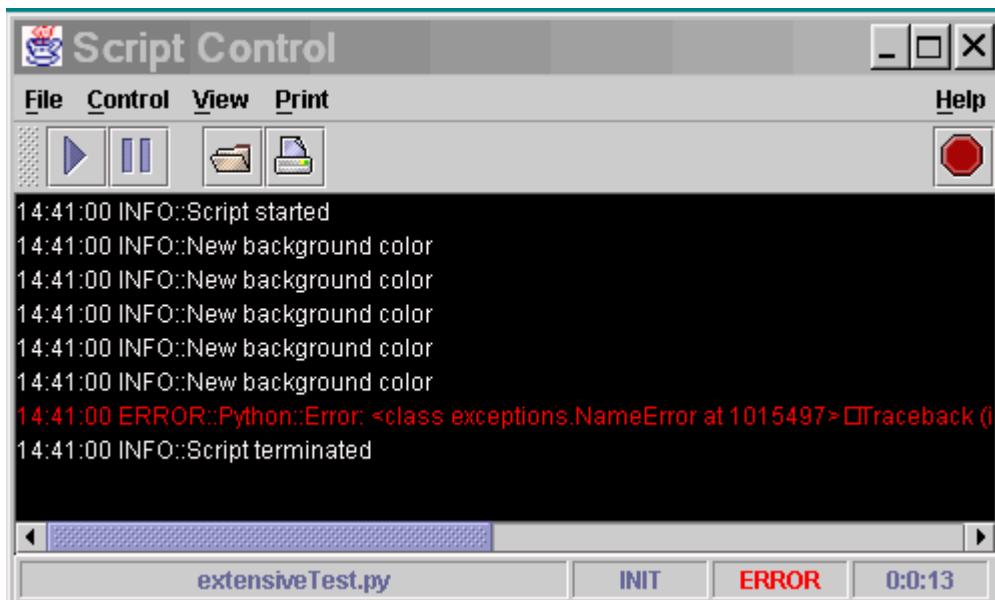
In the previous article I have explained why to use scripting and introduced Python, a powerful scripting language. Python, or rather its Java version Jython (www.jython.org), has been selected as a scripting language for the EMS project. Consequently, a scripting package has been developed to enable users to customize applications to suit their needs, ease their routine tasks, and automate tests. Repetitive tasks and complex procedures can now be simplified by providing scripts to handle them. With help of scripts, components can be "glued" together to form applications and measurements can be automated. In this installment we will focus on the use of Jython to control execution of EMS applications and will discuss the EMS scripting package.

•EMS scripting package

The scripting package contains two components: *ScriptInterpreter*, and *ScriptControlPanel* .

The *ScriptInterpreter* component is responsible for interpreting Jython scripts. In response to a START control event, it invokes an interpreter to process a script specified by the *scriptName* property. The invoked interpreter runs in a separate thread to allow for concurrent execution of the EMS framework and the script. The *ScriptInterpreter* component serves also as an intermediary between the script and other components. It sends control, property, and exception events on behalf of the script and receives replies.

The user can conveniently run scripts using *ScriptControlPanel*, which is a GUI component to monitor and control execution of scripts. With help of this component, the user can select a script for execution then start, pause, or abort the execution. *ScriptControlPanel* shows in its status bar the name of the interpreted script, state and error status of the script interpreter, and the elapsed execution time.



A script interpreter component has to be properly "wired" with a GUI component to achieve proper results. Control and property events from *ScriptControlPanel* have to be sent to *ScriptInterpreter* whereas exceptions have to be sent the opposite way:

```

<component id="Panel"
           class="ems.core.scripting.ScriptControlPanel">
    <property name="title" value="Script Control"/>
    <property name="XPosition" value="750"/>
    <property name="YPosition" value="610"/>
    <property name="capacity" value="100"/>
    <property name="width" value="500"/>
    <property name="height" value="350"/>
</component>

<component id="Interpreter"
           class="ems.core.scripting.ScriptInterpreter">
    <property name="controlDebugEnabled" value="true"/>
    <property name="propertyDebugEnabled" value="true"/>
    <property name="scriptName"
              value = "//V:/scripting/tests/test.py"/>
</component>
<route type="Exception"
       origin="Interpreter"
       destination="Panel"/>
<route type="Control"
       origin="Panel"
       destination="Interpreter"/>
<route type="Property"
       origin="Panel"
       destination="Interpreter"/>
```

●Basics of running EMS applications

Before we start exploring how to write scripts, let us briefly repeat the basics of running applications in EMS. EMS applications consist of communicating components. Typically, EMS components are data driven, which means that they automatically start processing data when they receive them. Of course, it happens only if a component is in an appropriate state, such as the RUNNING state. To put the component into the required state, one needs to send an appropriate control event to it. To ensure a proper behavior of the component, its properties have to be set to the appropriate values too. It can be achieved by sending property events to the component.

In summary, EMS applications can be controlled by sending control and property events to various components and that is what we have to be able to do from inside a script.

A crash course in EMS scripting

Script's interface to EMS

Scripts communicate with EMS components via an intermediary object, *ScriptInterpreter*. Access to this object can be obtained in a script using the *obtainScriptInterpreter* factory method of *ScriptInterpreter*. The interpreter object provides convenient methods to:

- modify properties (*getProperty*, *setProperty* methods)
- send/receive controls (*sendControl*, *sendControlAndWait*, and *waitForControlReply* methods)
- report exceptions and significant events (*reportException*)
- validate component names used in a script (*validComponentName*)

For details, refer to the on-line javadoc documentation of *ScriptInterpreter* (wwwtsmtf.fnal.gov/ems_javadoc2.X).

A trivial script that announces its start and termination can look as follows:

```
# Test program in Python
from ems.core.scripting import ScriptInterpreter
from java.lang import Thread

interpreter = ScriptInterpreter.obtainScriptInterpreter()
interpreter.reportException(1, "Script started")
Thread.sleep(3000)
interpreter.reportException(1, "Script terminated")
```

Verifying component names

It is a good practice to verify component names at the very beginning of the script. It prevents the system from partially executing a script on an incompatible configuration of components. Component names can be verified individually by

invoking `validComponentName`. The following code checks if there is a component named "CurrentController":

```
if not component.validComponentName("CurrentController"):
    raise ScriptError(
        "Python::Error: No CurrentController component")
```

Sending control signals

Scripts can send control signals to components and wait for replies. The following example shows how to send a request to move and wait for its completion before proceeding any further.

```
component.sendControlAndWait("MotionController",
                             "moveAbsolute", 10)
```

The script can also send a request, continue processing and wait for the completion of the requested action at some other point, by using separate calls to send a control signal and to wait for a reply:

```
component.sendControl("MotionController", "moveAbsolute")
# do some processing here ...
component.waitForControlReply("moveAbsolute", 10)
```

Modifying component properties

One can modify component properties by invoking the `setProperty` method. For example, code to set up default ramp parameters may look as follows:

```
component.setProperty("CurrentController",
                      "rampSpeed", Double(10.))
component.setProperty("CurrentController",
                      "acceleration", Double(10.))
```

One can examine properties by calling the `getProperty` and `getPropertyReplyByName` methods. The former method sends a get property request to a specified component, whereas the latter method retrieves the result of this request, as shown below:

```
# Check the component's state -----
component.getProperty("CurrentController", "state");
state = component.getPropertyReplyByName("state");
component.reportException(1, "CurrentController's state: "
                           + state)
```

Calls to `getProperty` and `getPropertyReplyByName` can be separated by some other code to allow for asynchronous execution. Before calling `getPropertyReplyByName`, one can make sure that the reply has already been received by calling the `isPropertyReplyAvailable` method.

Parameterized scripts

Sometimes the same algorithm has to be repeated with a different set of parameters. In such a case, it is a good idea to provide a separate file with parameters rather than to create a new version of the same script. The *getParameterFileName* method returns the name of the parameter file to use. Actual code may look like this:

```
paramFile = open(component.getParameterFileName(), 'r')
s = paramFile.readlines()
paramFile.close()
if s is None:
    raise ScriptError("Python::Error: "
                      + component.getParameterFileName()
                      + " parameter file not found")
```

The use of the parameters from the file depends on a particular script. One way to do it is shown in an example included below.

Exception handling

Scripts can announce some events to other components, such as error monitors. It can be done by calling the *reportException* method, which accepts two parameters. The first parameter defines the type of an event to be announced (error (3), warning (2), or event (1)), whereas the second parameter contains an event description string.

In order to capture exceptions generated by a Python script, one should enclose his/her code in a try block. The following Python code shows how to handle exceptions and can be used as a template for developing new scripts:

```
# Import classes -----
import sys
import org.python.core.PyString
from ems.core.scripting import ScriptInterpreter
from ems.core.scripting import InterpreterException

# Define an error class -----
class ScriptError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return `self.value`

try:
    # Announce starting measurement -----
    component = ScriptInterpreter.obtainScriptInterpreter()
    component.reportException(1, "Measurement started")

    # Verify component names -----
    if not component.validComponentName("CurrentController"):
```

```

        raise ScriptError(
            "Python::Error: No CurrentController component")
    # Processing -----
# Handle exceptions -----
except ScriptError, e:
    component.reportException(3, e.value)

except InterpreterException, e:
    component.reportException(3, e.toString())

except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
    component.reportException(3, "I/O error:" + strerror)

except:
    type, value, traceback = sys.exc_info()[:3]
    component.reportException(3, "Python::Error: " \
        + type.toString() + "\n" + traceback.dumpStack())

# Terminate -----
component.reportException(1, "Measurement terminated")

```

Example script

Let us now put it all together and examine a complete script that iterates through given sets of currents and positions. Both the currents and the positions are defined in a separate parameter file.

```

# Example script in Jython
# Author: Jerzy Nogiec
# Date: March 6, 2002
# $Revision$ $Date$

# Import classes -----
import sys
import org.python.core.PyString
from java.lang import Double
from ems.core.scripting import ScriptInterpreter
from ems.core.scripting import InterpreterException

# Define an error class -----
class ScriptError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return `self.value`

try:
    # Announce starting measurement -----
    component = ScriptInterpreter.obtainScriptInterpreter()
    component.reportException(1, "Measurement started")

    # Verify component names -----
    if not component.validComponentName("CurrentController"):
        raise ScriptError(>

```

```

    "Python::Error: No CurrentController component")
if not component.validComponentName("MotionController"):
    raise ScriptError(
        "Python::Error: No MotionController component")

# Initialize components -----
component.sendControl("CurrentController", "init")
component.sendControl("MotionController", "init")
component.sendControl("CurrentController", "start")
component.sendControl("MotionController", "start")

# Setup parameters -----
component.setProperty("CurrentController",
                      "rampSpeed", Double(10.))
component.setProperty("CurrentController",
                      "acceleration", Double(10.))

# Read positions and currents from a parameter file ---
paramFile = open(component.getParameterFileName(), 'r')
s = paramFile.readlines()
paramFile.close()
if s is None:
    raise ScriptError("Python::Error: "
                      + component.getParameterFileName()
                      + " parameter file not found")

# Ramp and move -----
for i in range(len(s)):
    if s[i][0] == '#':
        continue
    list = s[i].split()
    current = list[0]
    positions = list[1:]
    # Set current
    component.setProperty("CurrentController",
                          "requestedCurrent",
                          Double(current))
    component.sendControlAndWait("CurrentController",
                                 "ramp", 60)
    component.reportException(1,
                             "Current= " + current + "[A]")
    # Iterate through positions
    for j in range(len(positions)):
        position = positions[j]
        component.setProperty("MotionController",
                              "requestedPosition",
                              Double(position))
        component.sendControl("MotionController",
                              "moveAbsolute")
        component.waitForControlReply("moveAbsolute",
                                      10)
        component.reportException(1,
                                 "Position= " + position + "[m]")

    # Set current to 0 Amps and position to HOME -----
    component.setProperty("CurrentController",
                          "requestedCurrent",

```

```

        Double(0.))
component.sendControlAndWait("CurrentController",
                            "ramp", 60)
component.reportException(1, "Current= 0 [A]")
component.sendControlAndWait("MotionController",
                            "moveHome", 10)
component.reportException(1, "Position= HOME")

# Handle exceptions -----
except ScriptError, e:
    component.reportException(3, e.value)

except InterpreterException, e:
    component.reportException(3, e.toString())

except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
    component.reportException(3, "I/O error:"
                            + strerror)

except:
    type, value, traceback = sys.exc_info()[:3]
    component.reportException(3, "Python::Error: " \
                            + type.toString() + "\n"
                            + traceback.dumpStack())

# Terminate -----
component.reportException(1, "Measurement terminated")

```

The parameter file contains the following data:

```

# For each current iterate through a set of positions.
# Fields are separated by spaces.
# current [A]      positions [m]
50          1.0 2.0 3.0 4.0 5.0
100         1.0 2.0 3.0 4.0
200         1.0 2.0 3.0
300         1.0 2.0
400         1.0

```

Resources

If you would like to experiment with Jython outside of EMS, go to the Jython project homepage (www.jython.org), get a copy of the latest release, and install it. Run the java interpreter with the *jython.jar* file in your CLASSPATH. Use the batch file generated by Jython installer and modify it if necessary.

- *Python Programming in the JVM*, JDJ, Vol.5, Issue 3 <http://jdj.sys-con.com/read/36588.htm>
- EMS project page, <http://sdsg.fnal.gov/emsweb>
- Python project page, <http://www.python.org>
- Jython project page, <http://www.jython.org>

